

In this tutorial we will build a example query pipeline. The end goal is a application that takes as input a query image with location data and returns a matching to a closest database image.

## Step 1: Preprocess Database Images

First, you need to create the image database. You'll first want to process the database images into a certain format described below. All the resultant images should be put in a separate directory. If the source images are panoramas they should be sliced into overlapping images (example in `desphericalize.py`).

Format specification: given database image, lat, lon, create:

```
lat,lon-id.pgm
lat,lon-id.sift.txt
lat,lon-id.info
```

where lat, lon are formatted as floating point representations of latitude and longitude of the photo. The id field is a 4-digit decimal field to distinguish between images having the same latitude and longitude. It is commonly used to denote the view angle in a panorama slice.

Here's a step-by-step run-through using a few example images.

First copy the source images and the .info files into a folder. The .info files aren't really needed except for pose estimation and image drawing, so we provide them in this tutorial. There's an example on how to create them in `desphericalize.py`

```
$ mkdir testdb
$ cp tutorial/example_db_images/* testdb
$ ls testdb
  37.875507,-122.264883-0000.info  37.875507,-122.264883-0002.info
  37.875507,-122.264883-0001.info  37.875507,-122.264883-0003.info
  37.875507,-122.264883-0000.png  37.875507,-122.264883-0002.png
  37.875507,-122.264883-0001.png  37.875507,-122.264883-0003.png
```

Convert the images to pgm using the client util library.

```
$ python
>>> import client, glob
>>> for path in glob.glob('testdb/*.png'):
...     client.preprocess_image(path, path[:-4] + '.pgm', width=300,
height=225)
```

Extract SIFT features.

```
>>> for path in glob.glob('testdb/*.pgm'):
...     client.extract_features(path)
```

The necessary database files have now been created.

## Step 2: Specify Cell Structure

A query pipeline must have a db consisting of cells of a fixed radius and geometric configuration. Choose a suitable geometric configuration [see publications], then use the cell util library to configure the database cell structure.

Assuming we choose radius=distance=236.6m, build the cells. You want to tell util.mkcells the top-left corner of a box and the box's side length. See util.py for exact details on cell generation.

```
$ mkdir testdb/cells-236.6
$ python
>>> import util
>>> util.makecells(
...     lat=37.879,
...     lon=-122.270,
...     length=1000,
...     inputDir='testdb',
...     distance=236.6,
...     radius=236.6,
...     outputDir='testdb/cells-236.6/')
>>> exit()
$ ls testdb/cells-236.6
37.8753185464,-122.264614767  37.8771592272,-122.263268249
37.875318577,-122.2673074    37.8771592578,-122.265960949
```

## Step 3: Write code to drive the query

Now the cell db is fully set up, we can construct a driver to run our image query. We'll use the example query image tutorial/example\_query\_images/query1.png

First, we should set up a query context that points to the database we just set up. Create a file called 'tutorialTest.py' with the following class definition:

```
from context import _Context
import os

class TestContext(_Context):
    def __init__(self):
        super(TestContext, self).__init__()
        self.QUERY = 'SingleImageTest'

    @property
    def dbdump(self):
```

```

        return 'testdb'

    @property
    def dbdir(self):
        return 'testdb/cells-236.6'

    @property
    def matchdir(self):
        p = 'test_matches'
        if not os.path.exists(p):
            os.mkdir(p)
        return p

```

The property definitions override just enough of the `_Context` class to point to our newly created test database. See `context.py` for all the parameters that can be set via the context object.

To proceed, we should process the query image and then use the `system.match` function to match it against the query database. We add the following to a new file `tutorialTest.py`

```

import client
import system

image = 'tutorial/example_query_images/query1.png'

client.preprocess_image(image, image[:-4] + '.pgm', width=200,
height=200)
client.extract_features(image[:-4] + '.pgm')

C = TestContext()
C.check()

Q = _Query()
Q.jpgpath = image
Q.siftpath = os.path.splitext(image)[0] + "sift.txt"
Q.setSensorCoord(37.875507, -122.264883)
Q.check()

stats, matchedimg, matches, ranked = system.match(C, Q)

print "Matched db image ", matchedimg
print "Visualization in ", C.resultsdir

```

When we run `tutorialTest.py` for the first time, the query system will find the sift features and build the cell indexes. Upon further runs, queries will run faster using the prebuilt indexes. A successful first run will look like the following.

```

ericl@GORGAN:~/src$ python tutorialTest.py
[10680] 8:39:52 - --> Image conversion: 0.0179250240326s
Finding keypoints...
256 keypoints found.

```

```
[10680] 8:39:52 - --> Feature extraction: 0.229584932327s
[10680] 8:39:52 - Using 4 cells
[10680] 8:39:52 - I think we have enough memory for 7 threads
[10680] 8:39:52 - finding sift files
[10680] 8:39:52 - finding sift files
[10680] 8:39:52 - finding sift files
[10680] 8:39:52 - finding sift files
[10680] 8:39:52 - 123/1192 features read
[10680] 8:39:52 - 123/1192 features read
[10680] 8:39:52 - 255/1192 features read
[10680] 8:39:52 - 456/1192 features read
[10680] 8:39:53 - saving features... [37.875318577,-122.2673074]
[10680] 8:39:53 - saving features... [37.8753185464,-122.264614767]
[10680] 8:39:53 - saving features... [37.8771592578,-122.265960949]
[10680] 8:39:53 - saving features... [37.8771592272,-122.263268249]
[10680] 8:39:53 - creating 37.875318577,-122.2673074-kdtree1.uint8.index
[10680] 8:39:53 - creating 37.8771592578,-122.2659609-kdtree1.uint8.index
[10680] 8:39:53 - creating 37.8771592272,-122.2632682-kdtree1.uint8.index
[10680] 8:39:53 - creating 37.8753185464,-122.2646147-kdtree1.uint8.index
[10680] 8:39:53 - voting with method matchonce
[10680] 8:39:53 - accepted 150/256 votes
[10680] 8:39:53 - discarded 46 vote collisions
[10680] 8:39:53 - voting with method matchonce
[10680] 8:39:53 - voting with method matchonce
[10680] 8:39:53 - voting with method matchonce
[10680] 8:39:53 - accepted 150/256 votes
[10680] 8:39:53 - discarded 46 vote collisions
[10680] 8:39:53 - accepted 150/256 votes
[10680] 8:39:53 - discarded 46 vote collisions
[10680] 8:39:53 - accepted 150/256 votes
[10680] 8:39:53 - discarded 46 vote collisions
[10680] 8:39:53 - stopped after filtering 1

Matched db image 37.875507,-122.264883-0002
Visualization in /home/ericl/topmatches
```

That's it - all the match information has been output. For more complex examples, such of a pipeline set up to do batch processing, see `querySystem.py` or the more detailed documentation in the tutorial/Documentation folder.